A Framework for Load Balancing of Tensor Contraction Expressions via Dynamic Task Partitioning

Pai-Wei Lai*, Kevin Stock*, Samyam Rajbhandari*,
Sriram Krishnamoorthy†, P. Sadayappan*

*Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43221
{laip,stockk,rajbhand,saday}@cse.ohio-state.edu

†Computational Sciences and Mathematics Division
Pacific Northwest National Laboratory
Richland, WA 99352
sriram@pnnl.gov

ABSTRACT

In this paper, we introduce the Dynamic Load-balanced Tensor Contractions (DLTC), a domain-specific library for efficient task parallel execution of tensor contraction expressions, a class of computation encountered in quantum chemistry and physics. Our framework decomposes each contraction into smaller unit of tasks, represented by an abstraction referred to as *iterators*. We exploit an extra level of parallelism by having tasks across independent contractions executed concurrently through a dynamic load balancing runtime. We demonstrate the improved performance, scalability, and flexibility for the computation of tensor contraction expressions on parallel computers using examples from Coupled Cluster (CC) methods.

Keywords

Tensor contraction, domain-specific language, dynamic load balancing, task scheduling library

1. INTRODUCTION

Tensor contractions are generalized higher dimensional matrix multiplication operations that often arise in many-body methods in quantum chemistry, such as the Coupled Cluster (CC) methods [8]. Rapid development of high performance implementations of new and sophisticated methods is crucial for progressing the science. As manual implementation of these calculations is not feasible, automated approaches, such as the Tensor Contraction Engine (TCE) [2, 4, 13] exist and enable chemists to specify expressions in a high-level form from which parallel programs are automatically synthesized. The production chemistry software NWChem [23] contains more than two million lines of code generated by TCE.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13 November 17-21, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 $15.00
http://dx.doi.org/10.1145/2503210.2503290 .

TCE utilizes a blocked representation to exploit symmetry and sparsity of tensors. Only blocks of a tensor containing non-zero elements that cannot be derived by transposing other symmetric blocks are stored in Global Arrays (GA) [19]. A tensor contraction is translated into a subroutine of nested loops that contracts individual blocks. The nested loops are parallelized over multiple processors with a dynamic load balancing scheme. In the past, exploiting intra-contraction parallelism was sufficient. However, recent work [14] has shown scalability issues, such as small contractions cannot strongly scale to the size of modern supercomputers.

TCE is a compiler that translates high-level expressions representing tensor contractions into FORTRAN source code with fixed parameters. The large volume of code for high excitation methods causes long compilation times and inconveniences for code distribution and manipulation. In addition, further manual optimizations and performance tunings to the generated code have brought about divergence between TCE's output and the implementation of some methods used in NWChem.

In reaction to these issues, this paper presents Dynamic Load-balanced Tensor Contractions (DLTC), a light weight library for task parallel execution of tensor contraction expressions. DLTC exploits inter-contraction parallelism to maintain load balancing among processors. We are not aware of previous efforts that address automatic parallelization of multiple contractions. DLTC provides domain-specific primitives that enable dynamic decomposition of tensor contractions into fine-grained units of work. These tasks subsequently are scheduled, executed, and dynamically load balanced via a task execution runtime. DLTC allows the user to express arbitrary tensor contractions in a higher abstraction, while the details of intra- and inter-parallelization, data distribution, and load balancing remain hidden. DLTC is evaluated using CC methods and demonstrates improved performance and scalability in comparison with other state-of-art implementations.

The principal contributions of DLTC include:

- DLTC is a sustainable library for arbitrary tensor contraction expressions. The library approach improves programming productivity and reduces code volume compared with previous work of TCE.
- DLTC provides a domain-specific language for users

to express tensor contractions, which allows automatic task decomposition, data distribution, and dynamically load-balanced execution.

- By executing tasks across independent contractions concurrently, DLTC eliminates many synchronization points enforced by TCE, leading to higher processor utilization.
- DLTC can overlap computation and communication by using two sets of buffers on each process, further reducing the total execution time.

The rest of the paper is organized as follows: §2 covers the background and works related to this paper. §3 describes the problem being addressed. §4 describes the framework of DLTC. Experimental results are reported in §5, and §6 concludes the paper.

2. BACKGROUND AND RELATED WORK

We provide a brief introduction of the tensor contraction expressions and previous research work, partitioned global address space (PGAS) programming models, and task parallel execution models, which DLTC is built on.

2.1 Tensor Contraction Expressions

Consider the following example of a tensor contraction expression:

$$R[a,b,i,j] = T[a,b,k,l] * V[k,l,i,j]$$

This expression involves two input tensors V and T, which contract to produce an output tensor R. The concept for a tensor is similar to that of a matrix, except a tensor may have an arbitrary number of indices (four in this example). The indices a, b, i, and j are called *external* indices that come from either one of the input tensors and appear in the output tensor. The indices k and l are *contracted*, or *internal* indices, that are common to both input tensors but vanish after the contraction. Each index may have a different range and appear in any order.

Tensor contraction expressions arise in several applications among computational chemistry research areas. In particular, we focus on the tensor contractions that occur in the context of CC theory implementation, an accurate scheme widely employed to solve the Schrödinger equation. Efficient computations of tensor contraction expressions have been extensively studied [2, 4, 13]. Previous efforts have been made to reduce the number of operations in chains of tensor contraction expressions [11, 12, 15], minimize their memory requirement [5], trade off increased computation cost to reduce space [6], and minimize communication [21].

2.2 PGAS Programming Models

The PGAS programming model enables efficient and scalable implementations of task parallel programs on distributed memory machines by providing a global view of physically distributed data along with one-sided communication mechanisms. In our implementation, we focus on using GA [19], a portable library built upon the Aggregate Remote Memory Copy Interface (ARMCI) [18]. GA provides a convenient model for distributed multidimensional arrays, and ARMCI provides a rich set of efficient one-sided communication operations. Under this model, a tensor contraction computation can be implemented as several *get-compute-put* tasks, where data is first fetched into a local buffer, computed locally, and then copied back to a remote location after computation. GA is available on many computing platforms and employed in several production applications, including NWChem [23].

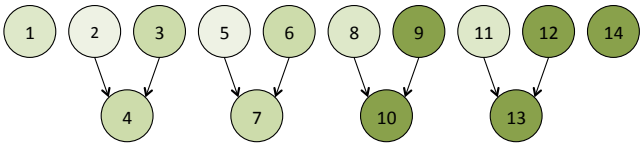

Figure 1: DAG representation of the CCD T2 method.

2.3 Task Parallel Execution Models

In a task parallel execution model, parallel computation is expressed as a set of dynamic tasks. A task is the basic unit of work and represented by its *task descriptor*, which provides references to the location of input and output data in the global address space. The runtime system creates local task queues that are split into a reserved segment and a shared segment. The reserved segment allows lock-free local access, and the shared segment allows one-sided access for remote task stealing. Because the tasks defined in DLTC operate on global address data, they can be executed at any processor and enable the runtime system to perform automatic task scheduling and dynamic load balancing. We report experimental results of DLTC using a GA-based library, Shared Collections of Task Objects (Scioto) [9, 10], and a message passing interface (MPI)-based Task Scheduling Library (TASCEL) [16].

3. PROBLEM STATEMENT

A typical CC method consists of tens to hundreds of tensor expressions. The input to our framework is a set of binary tensor expressions for which the total number of operations has been reduced through algebraic transformation [11, 12, 15]. To better explain our framework, Table 1 lists the input expressions of the Coupled Cluster Doubles (CCD) T2 method. An index name beginning with either a letter p or letter h designates that it is either in a virtual orbital range V or an occupied orbital range O, respectively. In practice, V is usually greater than O by a factor of tens to hundreds. DLTC is capable of executing arbitrary ranges of indices. However, for conciseness throughout this paper, we remove the coefficient in the expressions and use a problem size N to represent both O and V, where $N = O = V$ without loss of generality. The computational complexity of each expression and the additional global memory requirement for intermediate tensors W, X, Y, and Z also are shown.

As shown in Figure 1, the set of input expressions can be modeled as a directed acyclic graph (DAG) $G(V, E)$. Each vertex, or node, represents either a binary tensor contraction (C+=AB) or a unary tensor addition (X+=Y). An edge stands for the constraint that a certain node must be computed before another. Darker nodes have a higher arithmetic complexity.

TCE converts abstract expressions into parallel programs by first finding a topological ordering of the input DAG based on the dependencies of the contractions. Each node is translated into a separate subroutine program with parallelized loops, and nodes cannot be executed concurrently.

Node	Expression	Type	Complexity	Intermediate Memory
1	$R[p3,p4,h1,h2]+=V_1[p3,p4,h1,h2]$	ADD	$O(N^4)$	-
2	$W[h5,h1]+=F_1[h5,h1]$	ADD	$O(N^2)$	N^2
3	$W[h5,h1]+=T[p6,p7,h1,h8]*V_2[h5,h8,p6,p7]$	MUL	$O(N^5)$	N^2
4	$R[p3,p4,h1,h2]+=T[p3,p4,h1,h5]*W[h5,h2]$	MUL	$O(N^5)$	N^2
5	$X[p3,p5]+=F_2[p3,p5]$	ADD	$O(N^2)$	N^2
6	$X[p3,p5]+=T[p3,p6,h7,h8]*V_2[h7,h8,p5,p6]$	MUL	$O(N^5)$	-
7	$R[p3,p4,h1,h2]+=T[p3,p5,h1,h2]*X[p4,p5]$	MUL	$O(N^5)$	N^2
8	$Y[h7,h9,h1,h2]+=V_3[h7,h9,h1,h2]$	ADD	$O(N^4)$	N^4
9	$Y[h7,h9,h1,h2]+=T[p5,p6,h1,h2]*V_2[h7,h9,p5,p6]$	MUL	$O(N^6)$	N^4
10	$R[p3,p4,h1,h2]+=T[p3,p4,h7,h9]*Y[h7,h9,h1,h2]$	MUL	$O(N^6)$	N^4
11	$Z[h6,p3,h1,p5]+=V_4[h6,p3,h1,p5]$	ADD	$O(N^4)$	N^4
12	$Z[h6,p3,h1,p5]+=T[p3,p7,h1,h8]*V_2[h6,h8,p5,p7]$	MUL	$O(N^6)$	N^4
13	$R[p3,p4,h1,h2]+=T[p3,p5,h1,h6]*Z[h6,p4,h2,p5]$	MUL	$O(N^6)$	N^4
14	$R[p3,p4,h1,h2]+=T[p5,p6,h1,h2]*V_5[p3,p4,p5,p6]$	MUL	$O(N^6)$	-

Table 1: Tensor contraction expressions of the CCD T2 method.

A shared global counter is used for assigning tasks and load balancing during each contraction.

One drawback of this approach is that all processors are required to participate in the computation of every single node, regardless of the diverse complexity of different expressions. The workload of one large contraction can be higher than a small contraction by an order of magnitude. For example, expression 14 in Table 1 has a workload of $O(N^6)$, while node 2 only has a workload of $O(N^2)$. While the computationally dominant contractions are well-optimized and scalable, the less expensive ones become the computational bottleneck due to communication and load imbalance. A noticeable overhead due to the use of a shared global counter also limits the scalability of this approach (reported in [20]).

We address this issue by constructing the input DAG into layers. Each layer consists of a group of independent nodes that can be executed concurrently to maximize the processor usage. Tasks are dynamically partitioned based on the problem size and available resources. A pipelining scheme is used to overlap communication with computation, and possible data replication is performed when sufficient memory is available.

4. DYNAMIC LOAD-BALANCED TENSOR CONTRACTIONS

In this section, we discuss the motivation and design in developing a task parallel execution library as a solution to calculate a set of tensor contraction expressions. We begin by introducing the basic components of DLTC. We present a baseline implementation of the approach employed in NWChem. Further optimizations are explained with simplified code snippets to describe the action taken by DLTC.

4.1 Domain-specific Primitives

While parallel implementations of chemistry methods can be derived from the TCE compiler, significant manual tuning is required to improve the performance and scalability of the generated code. By designing DLTC as a library, any modifications can be immediately effective in production code, and the amount of code for complicated methods is significantly reduced.

DLTC provides a small set of domain-specific primitives for the user to specify tensor expressions. The *tensor* class provides collective functions for creating and destroying a tensor, as well as non-collective functions for getting, putting, and accumulating a particular tensor block. DLTC employs a block sparse representation to handle symmetry and sparsity of tensors. DLTC supports two types of *expression* classes: binary tensor contractions and unary tensor additions.

Listing 1: Specifying a multiplication expression.

```
/* creating tensors */
int rW[] = {O,O};
int rT[] = {V,V,O,O};
int rV[] = {O,O,V,V};
Tensor W = Tensor(2,rW);
Tensor T = Tensor(4,rT);
Tensor V = Tensor(4,rV2);

/* specifying a contraction */
int iW[] = {H5,H1};
int iT[] = {P6,P7,H1,H8};
int iV[] = {H5,H8,P6,P7};
W.setIndexName(2,iW);
T.setIndexName(4,iT);
V.setIndexName(4,iV2);
Mult M = Mult(W,T,V,1.0);
```

Listing 1 is a code snippet that shows an example of how these primitives are used to specify the binary contraction of node 3 in Table 1. The mapping of high-level tensor contraction expressions into domain-specific primitives is achieved through the development of an automated code generator.

DLTC tensors are created by specifying the numbers and range of each dimension. For tensor expressions with symmetry or sparsity, tensor creation involves an additional step of checking whether a block has nonzero elements or is symmetric to an existing block. Currently DLTC employs a distribution of tensors where each tensor is distributed among all the processes using global address space. A contraction is specified by the index names of the input and output tensors and a coefficient. The specification of contraction determines which of the tensors' indices are internal and external. Possible index reordering is required to utilize matrix multiplication kernels for computation. For example, a contraction of:

$$C[i,j,m,n]+=A[i,j,k,l]*B[k,l,m,n]$$

Listing 2: A naive implementation of MatMul2D using three loops.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Listing 3: Creation of a single iterator for MatMul2D.

```
enum {I,J,K};
int id = {I,J,K};
int lb = {0,0,0};
int ub = {N,N,N};
Iterator It = Iterator(3,id,lb,ub);
```

essentially is the same as two-dimensional (2D) matrix multiplication, while A in the following form of contraction on the same tensors requires a transpose among index k and j.

$$C[i,j,m,n] += A[i,k,j,l] * B[k,l,m,n]$$

After the user declares the *tensors* and *expressions*, each expression is decomposed into computational tasks called *iterators*. An iterator loops over a subrange of the nested loops of block indices.

Consider the example of decomposing 2D matrix multiplication into iterators. Given the following expression:

$$C[i,j] += A[i,k] * B[k,j]$$

where C, A, and B are 2D matrices and the indices i, j, and k are in the same range N. A naive implementation for this expression is to write three nested loops, as shown in Listing 2.

Each iterator can be logically viewed as a subset of the computation and is used to represent a single task to be executed on any process. Given an iterator and its generating expression, a task function determine the on-node computation and processes the actual computation, which manages the transposes and double-precision General Matrix Multiply (DGEMM) calls.

Listing 3 displays an identical computation by creation of a single iterator that is capable of cycling through the entire range of all three indices. Listing 4 shows the MatMul2D task function, which takes this iterator as input and executes

Listing 4: Task function for 2D matrix multiplication.

```
MatMul2D(Iterator It) {
  IdsVal idv = IdsVal(3,id);
  while (It.next(idv)) {
    int i = idv.getVal(I);
    int j = idv.getVal(J);
    int k = idv.getVal(K);
    C[i][j] += A[i][k] * B[k][j];
  }
}
```

Listing 5: Creation of two iterators for MatMul2D.

```
int lb1 = {0,0,0};
int ub1 = {N,N,N/2};
int lb2 = {0,0,N/2+1};
int ub2 = {N,N,N};
Iterator It1 = Iterator(3,id,lb1,ub1);
Iterator It2 = Iterator(3,id,lb2,ub2);
```

Listing 6: Task function for multiplication expression.

```
Task_Mul_1(Mult M, Iterator It) {
  ... /* allocate temporary buffers */
  IdsVal idv;
  while (It.next(idv)) {
    M.tA.GET(idv,bufA);
    TRANSP(bufA,bufAt,M);
    M.tB.GET(idv,bufB);
    TRANSP(bufB,bufBt,M);
    DGEMM(bufAt,bufBt,bufCt);
    TRANSP(bufCt,bufC,M);
    M.tC.ACC(idv,bufC,M.coef);
  }
  ... /* free temporary buffers */
}
```

the actual computation. Within the task function, a loop calls the $next()$ function provided by the iterator as long as it returns true. The index names and their values within a particular iteration are stored in a table idv. Using this scheme, a computation can be partitioned arbitrarily into multiple iterators with different iteration spaces. Listing 5 is an example of breaking down the work into two iterators on the contracted index K. In DLTC, the computation is performed on blocks rather than single elements as shown in this MatMul2D example. Listing 6 illustrates the code snippet of a task function, which is the general algorithm of binary multiplication for dense tensors.

In the multiplication task function, local buffers $bufA$, $bufB$, and $bufC$ are allocated for temporary storage of the input and output tensors. The input iterator cycles through a subrange of the block indices of the input expression with the index-value mapping stored in idv. The information provided by the iterator and the expression determines which blocks to load and store the results. Each iteration can be considered as a dot product of two input blocks and an ac-

Listing 7: Task function for summation expression.

```
Task_Add_1(Summ S, Iterator It) {
  ... /* allocate temporary buffers */
  IdsVal idv;
  while (It.next(idv)) {
    S.tY.GET(idv,bufY);
    TRANSP(bufY,bufYt,S);
    S.tX.ACC(idv,bufYt,S.coef);
  }
  ... /* free temporary buffers */
}
```

Listing 8: NXTVAL scheme for task distribution.

```
Task_NXTVAL(Expr E, Iterator It) {
  ... /* allocate temporary buffers */
  int count = 0;
  int next  = NXTVAL();
  IdsVal idv;
  while (It.next(idv)) {
    if (next==count) {
      ... /* compute task */
      next = NXTVAL();
    }
    count += 1;
  }
  ... /* free temporary buffers */
}
```

cumulation of the result into an output block. Blocks are copied to the local buffer using one-sided GET operations implemented in *tensor*. A transposition, or index reordering, is required if the memory layout of this particular block is not suitable for computation. A vendor-provided Basic Linear Algebra Subroutine (BLAS) library performs matrix multiplication, and the calculated result is subsequently transposed (if needed) and accumulated to a remote location. Listing 7 illustrates a task function for the general algorithm of performing tensor addition.

4.2 Baseline Implementation

By only using DLTC to exploit intra-node parallelism, we create a baseline task distribution similar to that used by TCE, often referred to as the *NXTVAL* scheme. DLTC implements the *NXTVAL* scheme using the following procedure:

1. Create an iterator that cycles through the entire iteration space for a particular expression.
2. Replicate this iterator on every process.
3. Execute a modified task function with the *NXTVAL* function used inside the loop.

Listing 8 illustrates pseudo code of DLTC implementation for this model. All processes will execute this function at the same time with the replicated iterator mentioned in the previous steps. In this scheme, every process performs an atomic get-and-increment (*NXTVAL*) of a shared global counter to determine the next task it should execute by comparing it with a local counter. The return value of the global counter uniquely identifies the work to be done in the current task. All processes continue this procedure until no tasks remain for the current expression.

4.3 Exploiting Inter-Contraction Parallelism

To exploit parallelism across different contractions, we begin by dividing the nodes into several groups, so tasks in the same group are independent, i.e., can be executed concurrently. Partitioning the vertices of a DAG graph into layers, or levels, can be formalized as a variation of the *layering problem* [3, 22] described as follows:

Consider a DAG graph of tensor expressions $G = (V, E)$ with a set of vertices V and a set of directed edges E. Let $L = L_0, L_1, ..., L_h$ be a partition of the vertex set of G into $h >= 1$ subsets such that if a directed edge $E{=}u, v$ with u from L_j and v from L_i, then $i < j$. The partition process often is called *layering*, and the partitioned sets of $L_0, L_1, ..., L_h$ are referred to as *layers*, or *levels*, of G. In our problem, we seek a layering such that the number of layers, or height, is minimized, while the total extra memory required for intermediate tensors in each layer does not exceed a memory upper bound M. If we assume the memory available is sufficient for all of the intermediates, a longest-path algorithm can find the layering with minimal height in linear time [17]. The Coffman-Graham algorithm [7] is a greedy algorithm shown to be effective for solving a layering problem with a bounded number of vertices for each layer. The worst-case time complexity of the Coffman-Graham algorithm is $O(|V|^2)$. Rather than checking the number of vertices, we employ the following modified algorithm, which checks the total extra memory needed:

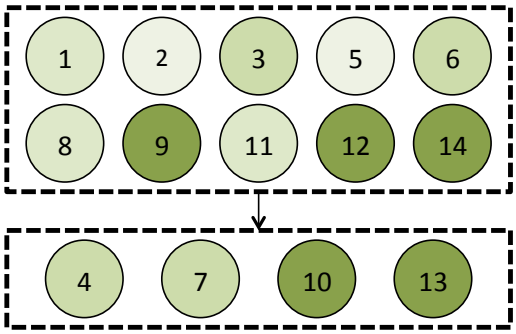

Figure 2: Layered DAG representation of the CCD T2 method.

1. Represent the partial order of the input DAG by removing all of the transitive edges.
2. Construct a topological ordering for the reduced graph.
3. For each vertex v in the reverse of the topological ordering, add v to the lowest level that is at least one level higher than the highest level of any outgoing vertices of v and does not exceed the extra memory requirement M.

By layering the DAG into a minimal number of levels, many of the synchronization points that would otherwise exist between the expressions can be eliminated. Figure 2 shows the minimal height example of a layering for CCD T2. The expressions in the same level will be computed concurrently, preventing the system from inefficiently having all processes working on a single small contraction.

4.4 Task Decomposition

Given a particular expression, the TCE compiler approach produces a concrete translation from the expression into a subroutine program. The task partitioning is fixed, and the only controllable parameter is the block size set at the beginning of program execution that is shared across all expressions. This rigid characteristic is one drawback to using the compiler approach, because the performance of execution is highly dependent upon a proper block size selection.

Figure 3 shows the influence of block size selection using a static task partitioning scheme. The profiling is performed

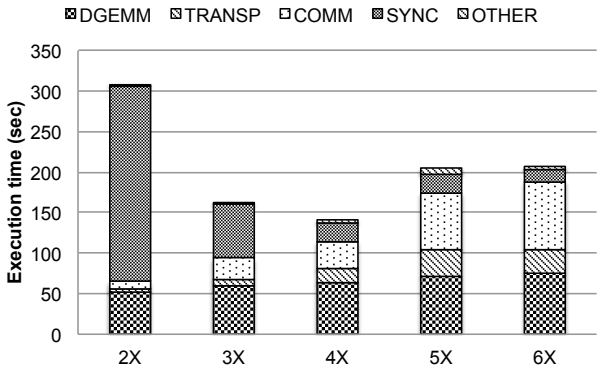

Figure 3: Influence of block size selection.

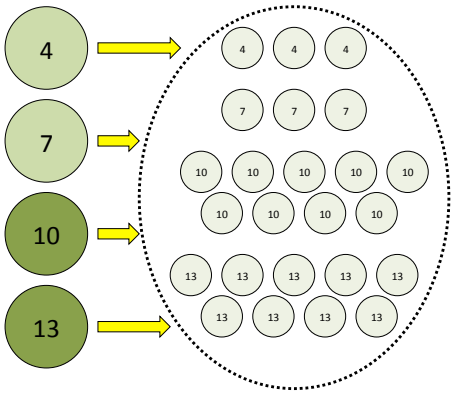

Figure 4: Task decomposition of a layer of nodes into a task pool.

on CCD method with a problem size $N = 100$, running on a 64 node (total of 512 cores) cluster using the baseline approach. The total execution time of each process is split into several categories: matrix multiplication (DGEMM), block transposition (TRANSP), communication (COMM), synchronization (SYNC), and other. We take the average of the performance over all processors and present the breakdown in a stacked bar graph. The x-axis shows the ratio of N over the block width chosen, and the y-axis shows the execution time in seconds. While choosing a larger block size requires less communication time, the insufficient number of tasks leads to poor load balance and incurs extremely high synchronization times. Choosing a smaller block size can improve load balance. However, the overhead for both communication and block transposition are increased due to a larger number of smaller tasks and blocks. The best execution time is obtained by a careful selection of block size, which compromises between the overhead of communication and synchronization. Even for the best execution time, the synchronization time is nearly 17% of the total execution time.

In DLTC, the granularity of a single task is defined by both the iterator and chosen block size. As shown in Figure 4, rather than a static task decomposition scheme for all expressions, DLTC attempts to split all of the nodes in a level so each task has a similar amount of work. DLTC determines the task granularity of each node at runtime based

Listing 9: Task decomposition for an expression.

```
Decomp(Expr E, TaskPool tp) {
  std::vector<Iterator> Its;
  if E.cost < T1
    ... /* create one tasks for E */
  else if E.cost < T2
    ... /* create external indices
           level tasks */
  else
    ... /* create summation indices
           level tasks */

  addTask(Its,tp);
}
```

on the information provided from the expression and problem size. This dynamic scheme of partitioning tasks helps to maintain better load balancing. The code in Listing 9 is a simple partitioning example that allows three different kinds of iterator granularity to breakdown a given expression. Based on the operation cost from the expression, the thresholds $T1$ and $T2$ determine whether or not the entire expression forms a single iterator, decomposed at the external or internal indices level. Task partitioning at the external level produces an iterator that iterates over internal indices. If decomposed at internal indices level, each iterator essentially is one single block-block computation. More complex partitioning schemes can be developed with architecture-specific or empirically driven performance models, which requires further investigation and profiling of test executions.

4.5 Task Distribution and Execution

In the task pool programming model, computation is expressed as a set of tasks to be executed in parallel. The task descriptor is used to obtain the address of inputs and outputs in the global address space. In DLTC, the iterators serve as the task descriptor to identify the task. First, we create a separate task pool for each layer. The iterators produced from the nodes in the same layer goes into the same task pool. The tasks in each task pool are distributed using a round-robin fashion and executed concurrently, without taking locality into account. The task parallel execution runtime takes care of work stealing when a process is out of work. Few synchronization points are needed between layers, and no global locks are used for task distribution.

4.6 Pipelining

With the PGAS model's support for one-sided, non-blocking communication, we can overlap data movement with computation. Listing 10 shows a double buffering technique used for hiding communication. By allocating two set of local memory buffers, a process can fetch the first set of buffers and then swap to process the second without waiting for the loading to be finished. A pointer p is used to indicate which set of buffers the current iteration process is on. This can be repeated until the last iteration. No adaption to the layering process is required to support pipelining. Clearly, this technique will finish the task faster because the time originally spent on waiting now can be used to process computation on a separate set of buffers. Although this technique will re-

Listing 10: Task function of multiplication expression with double buffering.

```
Task_Mul_2(Mult M, Iterator It) {
  IdsVal idv[2];
  ... /* allocate 2X temporary buffers */
  bool p = 0; // pointer to current buffer
  bool first = true;
  while (It.next(idv[p])) {
    M.tA.NBGET(idv[p],bufA[p],p);
    M.tB.NBGET(idv[p],bufB[p],p);
    p = !p; // toggle
    if (first) first=false;
    else {
      M.tA.NBWAIT(p);
      M.tB.NBWAIT(p);
      TRANSP(bufA[p],bufAt[p],M);
      TRANSP(bufB[p],bufBt[p],M);
      DGEMM(bufAt[p],bufBt[p],bufCt[p]);
      TRANSP(bufCt[p],bufC[p],M);
      M.tC.ACC(idv[p],bufC[p],M.coef);
    }
  }
  if (!first) { // last iteration
    p = !p; // toggle
    M.tA.NBWAIT(p);
    M.tB.NBWAIT(p);
    TRANSP(bufA[p],bufAt[p],M);
    TRANSP(bufB[p],bufBt[p],M);
    DGEMM(bufAt[p],bufBt[p],bufCt[p]);
    TRANSP(bufCt[p],bufC[p],M);
    M.tC.ACC(idv[p],bufC[p],M.coef);
  }
  ... /* free 2X temporary buffers */
}
```

quire two times the amount of temporary buffers, the buffer sizes are relatively small comparing with the memory needed to store tensors.

5. EXPERIMENTAL EVALUATION

Performance evaluation was done on an InfiniBand cluster at The Ohio State University. The cluster is configured with 160 nodes, each node with two quad-core Intel Xeon CPU E5640@2.67GHz processors and 12GB memory per node. All nodes are connected to a single InfiniBand switch. The system runs the MVAPICH2-1.9 [1] MPI implementation and Redhat Linux Enterprise 5.4. DLTC is written in C/C++ and built using the Intel C Compiler 13. DLTC employs the Intel Math Kernel Library (MKL) for DGEMM, and the tensor transposition function is parallelized with OpenMP. We provide the performance result of DLTC using both GA (DLTC-G) and TASCEL (DLTC-T). DLTC-G uses the GA-5.0 [19] for PGAS model and GA-based Scioto [10] for task execution runtime. DLTC-T employs TASCEL-2.0 [16], a MPI-based library, for both PGAS model and task execution runtime. Different settings for performance tuning were tested, and the best time obtained for each implementation is reported.

The *NXTVAL* scheme from Listing 8 is presented as the baseline (BASE-G) in the experimental results. We use this implementation as a representative for NWChem's dynamic behavior on tensor contractions, because it employs the exact same task distribution and load balancing scheme based on GA. The difference is that the NWChem has explicit subroutine code (generated by TCE) for every expression, while BASE-G dynamically generates indices values from the input expressions on the fly. The overhead for dynamic parameter generation is within 3% of total execution time.

We compare the performance of DTLC against the implementations of CCD and Coupled Cluster Singles and Doubles (CCSD) methods in the Cyclops Tensor Framework (CTF) [21] and NWChem. CTF employs a cyclic data representation to preserve tensor structure and produces a regular tensor decomposition. In CTF, the execution is categorized into a computation phase using the Scalable Universal Matrix Multiplication, or SUMMA [24] algorithm and a data redistribution phase between contractions. CTF has been shown to be effective on computing single permutation symmetric contraction expressions, especially on torus clusters, due to its organized communication and topology mapping of tensor data onto processors. However, all of the processors still are enforced to be involved in every contraction, and redistribution of tensors is enforced between most contractions. We compare the performance of DLTC and CTF on dense tensor contractions as spin and spatial symmetries in tensors are not yet implemented in CTF. DLTC-G is designed to be numerically identical to CTF, while DLTC-T supports additional symmetries by implementing customized iterators, which are capable of cycling through indices with permutation, spin, and spartial symmetry properties employed in NWChem.

Although CC methods are computed iteratively, we focus on the performance of a single iteration of computation in this paper. In the first set of experiments, we compared the parallel performance of CCD and CCSD implementation in DLTC against the CTF and BASE-G. Figures 5 and 6 show the comparison results for CCD and CCSD computation as the problem size scales on 64 nodes (512 cores). For problem size of 100, DLTC-G executed CCD in 41 seconds and CCSD in 67 seconds, which is 2.02x and 2.44x faster than CTF, respectively. For smaller problems, the speedup is more evident due to the smaller percentage of time spent in computation. The execution time breakdown analysis on the problem size of 100 in CCSD calculation reports that both DLTC-G and CTF has nearly the same DGEMM time. However, it is 54% of total time for DLTC-G but only 18% for CTF, which may be due to the high overhead in synchronization and data redistribution. The benefit of topology-aware mapping in CTF also cannot be utilized on InfiniBand clusters, whereas ARMCI communication is heavily optimized for such networks. The breakdown time of DLTC-G is as follows: 54% for DGEMM, 20% for communication, and 18% for tensor transposition. The remaining 8% of the time is spent for dynamic load balancing. DLTC-T has a similar breakdown time as DLTC-G except a higher communication overhead due to the active-message-based communication functions provided by TASCEL.

The results of parallel strong scaling are shown in Figure 7 and Figure 8. We collect results by running a fixed problem size $N = 100$ as we scale the number of cores. The speedup is computed based on the best time obtained from DLTC-G on 16 nodes (128 cores). DLTC-G achieved a higher speedup over CTF — up to 2.58x for CCD and 3.03x for CCSD. Both graphs depict good improvement of scalability and also show that BASE-G scales poorly as the number of cores increases. We demonstrate that DLTC-G achieves scalability by applying an extra level of parallelism across

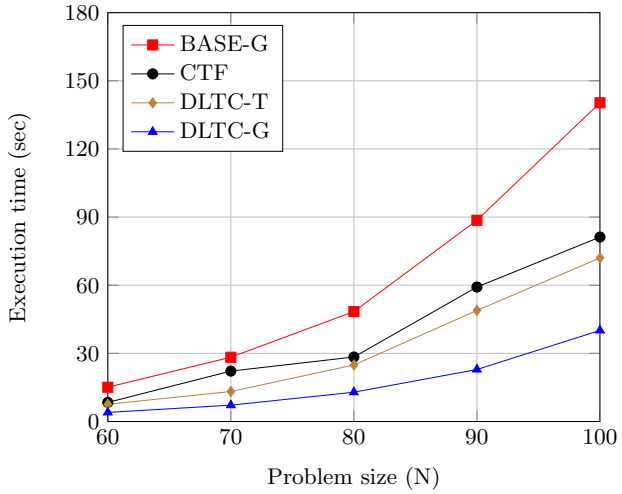

Figure 5: CCD running time (number of cores=512).

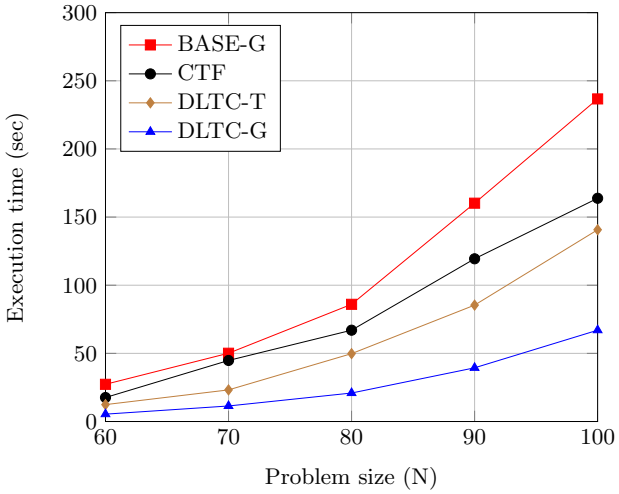

Figure 6: CCSD running time (number of cores=512).

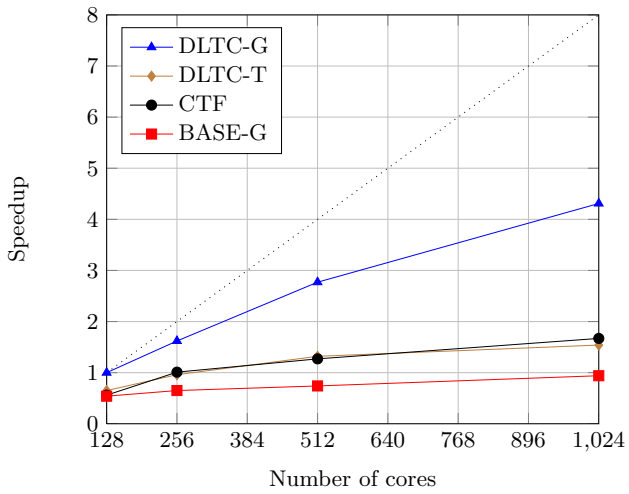

Figure 7: CCD strong scaling.

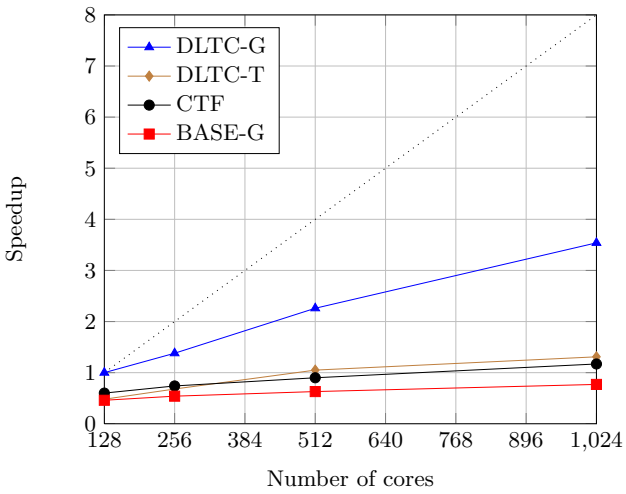

Figure 8: CCSD strong scaling.

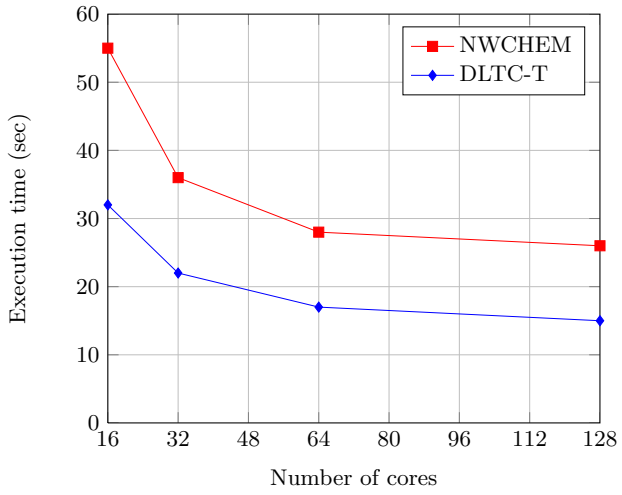

Figure 9: URACIL.

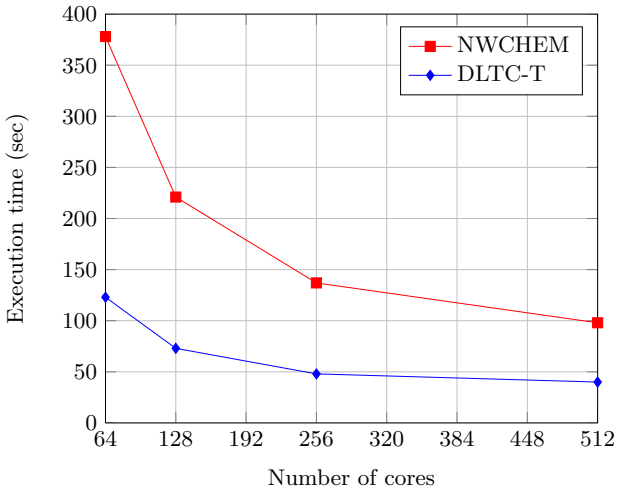

Figure 10: GFP.

Method	TCE	DLTC
CCSD T1	2775	304
CCSD T2	9561	471

Table 2: SLOC comparison between TCE and DLTC.

independent contractions.

The full CCSD code with permutation, spin, and spatial symmetries is generated and tested on two real molecule simulations: uracil and green fluorescent protein (GFP). The execution times of DLTC-T and NWChem are reported in Figure 9 and Figure 10, where DLTC-T completed the computation with 1.73x and 2.45x speedup over NWChem, respectively. Table 2 shows the number of source lines of code (SLOC) generated by NWChem TCE and DLTC-T for CCSD T1 and T2 expressions. Rather than generate a huge volume of code, there is a significant reduction because the computation is executed using the same task function.

Although DLTC still is under development, the framework is already capable of computing arbitrary tensor contraction expressions with no restriction on the number of dimensions. DLTC performance is expected to be enhanced by considering locality-aware task distribution and task stealing. Because each tensor is partitioned into a collection of dense blocks and distributed, it is feasible to map the tasks associated with the data they operate based on locality to further reduce communication overhead. DLTC also provides a sustainable abstraction for performance tuning.

6. CONCLUSION

This paper describes the DLTC, a framework which provides dynamic task partitioning of tensor contraction expressions. DLTC takes advantage of abstracting work into *iterator*s to map expressions into differing granularity tasks. The tasks are executed in a dynamic load-balanced fashion with pipelining to hide communication latency. We have evaluated DLTC in comparison with NWChem and CTF using CC methods on an InfiniBand cluster. Our work shows that DLTC achieves highly improved overall performance and suggests that viewing the entire set of expressions as a whole and exploiting parallelism across expressions offers better performance and scalability. The DLTC framework's library approach also significantly reduces source code size and eases programming and performance tuning for developers.

7. ACKNOWLEDGMENT

We thank Edgar Solomonik and Devin Matthews for making the CTF software available. We thank Wenjing Ma and Karol Kowalski for their help to run DLTC with actual inputs from NWChem. We also thank Dmitry Lyakh for providing the Coupled Cluster equations. This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award number 59193 and by Pacific Northwest National Laboratory (PNNL) under the eXtreme Scale Computing Initiative. PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830.

8. REFERENCES

[1] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

[2] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[3] Oliver Bastert and Christian Matuszewski. Layered drawings of digraphs. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs*, volume 2025 of *Lecture Notes in Computer Science*, pages 87–120. Springer Berlin Heidelberg, 2001.

[4] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276 –292, February 2005.

[5] D. Cociorva, J. W. Wilkins, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Loop optimization for a class of memory-constrained computations. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 103–113, New York, NY, USA, 2001. ACM.

[6] Daniel Cociorva, Gerald Baumgartner, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, Marcel Nooijen, David E. Bernholdt, and Robert Harrison. Space-time trade-off optimization for a class of electronic structure calculations. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 177–186, New York, NY, USA, 2002. ACM.

[7] E. G. Coffman, Jr. and R.L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[8] T. D. Crawford and H. F. Schaefer, III. An Introduction to Coupled Cluster Theory for Computational Chemists. In *Reviews in Computational Chemistry*, volume 14, pages 33–136. John Wiley and Sons, Inc., 2000.

[9] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 586–593, Washington, DC, USA, 2008. IEEE Computer Society.

[10] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[11] Albert Hartono, Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Marcel Nooijen, Gerald Baumgartner, David E. Bernholdt, Venkatesh Choppella, Russell M. Pitzer, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *Proceedings of the 6th International Conference on Computational Science - Volume Part I*, ICCS'06, pages 267–275, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] Albert Hartono, Alexander Sibiryakov, Marcel Nooijen, Gerald Baumgartner, David E. Bernholdt, So Hirata, Chi-Chung Lam, Russell M. Pitzer, J. Ramanujam, and P. Sadayappan. Automated operation minimization of tensor contraction expressions in electronic structure calculations. In *Proceedings of the 5th International Conference on Computational Science - Volume Part I*, ICCS'05, pages 155–164, Berlin, Heidelberg, 2005. Springer-Verlag.

[13] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[14] Karol Kowalski, Sriram Krishnamoorthy, Ryan M. Olson, Vinod Tipparaju, and E. Aprà. Scalable implementations of accurate excited-state coupled cluster theories: application of high-level methods to porphyrin-based systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 72:1–72:10, New York, NY, USA, 2011. ACM.

[15] Pai-Wei Lai, Huaijian Zhang, Samyam Rajbhandari, Edward Valeev, Karol Kowalski, and P. Sadayappan. Effective utilization of tensor symmetry in operation optimization of tensor contraction expressions. In *Proceedings of the 12th International Conference on Computational Science*, volume 9 of *ICCS'12*, pages 412 – 421, 2012.

[16] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 137–148, New York, NY, USA, 2012. ACM.

[17] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, volume 2 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1984.

[18] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. *International Journal High Performance Computing Applications*, 20(2):233–253, May 2006.

[19] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, May 2006.

[20] David Ozog, Sameer Shende, Allen Malony, Jeff R. Hammond, James Dinan, and Pavan Balaji. Inspector-executor load balancing algorithms for block-sparse tensor contractions. In *Proceedings of the 27th International Conference on Supercomputing*, ICS '13, pages 483–484, New York, NY, USA, 2013. ACM.

[21] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: reducing communication and eliminating load imbalance in massively parallel contractions. Technical Report UCB/EECS-2012-210, EECS Department, University of California, Berkeley, November 2012.

[22] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981.

[23] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. deJong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

[24] Robert A. van de Geijn and Jerrell Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency - Practice and Experience*, 9(4):255–274, 1997.